

Referencias y colecciones

lección9

En lecciones anteriores aprendiste como añadir tus propias acciones. Sin embargo, esto no es suficiente para personalizar del todo el comportamiento de tu aplicación, porque la interfaz de usuario generada, en concreto la interfaz de usuario para referencias y colecciones, tiene un comportamiento estándar que a veces no es el más conveniente.

Por fortuna, OpenXava proporciona muchas formas de personalizar el comportamiento de las referencias y colecciones. En esta lección aprenderás como hacer algunas de estas personalizaciones, y como esto añade valor a tu aplicación.

9.1 Refinar el comportamiento de las referencias

Posiblemente te hayas dado cuenta de que el módulo Order tiene un pequeño defecto: el usuario puede añadir cualquier factura que quiera al pedido actual, aunque el cliente de la factura sea diferente. Esto no es admisible. Arreglémoslo.

9.1.1 Las validaciones están bien, pero no son suficientes

El usuario sólo puede asociar un pedido a una factura si ambos, factura y pedido, pertenecen al mismo cliente. Esto es lógica de negocio específica de tu aplicación, por tanto el comportamiento estándar de OpenXava no lo resuelve.

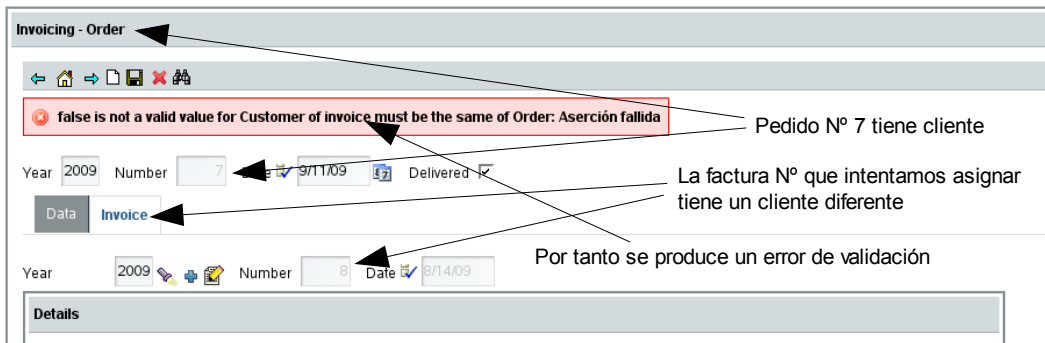


Figura 9.1 Error de validación cuando el cliente de la factura es incorrecto

Ya que esto es lógica de negocio la vamos a poner en la capa del modelo, es decir, en las entidades. Lo haremos añadiendo una validación. Así obtendrás el efecto de la figura 12.1.

Ya sabes como añadir esta validación a tu entidad Order. Se trata de añadir un método anotado con `@AssertTrue`. Puedes verlo en el listado 9.1.

Listado 9.1 Nuevo método de validación en la entidad Order

```
@AssertTrue // Este método tiene que devolver true para que este pedido sea válido
private boolean isCustomerOfInvoiceMustBeTheSame() {
    return invoice == null || // invoice es opcional
        invoice.getCustomer().getNumber()==getCustomer().getNumber();
}
```

Aquí comprobamos que el cliente de la factura es el mismo que el del pedido. Esto es suficiente para preservar la integridad de los datos, pero la validación sola es una opción bastante pobre desde el punto de vista del usuario.

9.1.2 Modificar los datos tabulares por defecto ayuda

Aunque la validación impide que el usuario pueda asignar una factura incorrecta a un pedido, lo tiene difícil a la hora de escoger una factura correcta. Porque cuando pulsa para buscar una factura, todas las facturas existentes se muestran, y lo que es todavía peor, la información del cliente no aparece en la lista. Fíjate en la figura 9.2.

When the user clicks to search for an invoice, the list does not show the customer's data.

	Year	Number	Date	Amount	Remarks	VAT %	Estimated profit	Base amount	V.A.T.	Total amount
Choose		7	8/14/09	863.04		16	86.30	744.00	119.04	863.04
Choose	2009	8	8/14/09	107.88		16	10.79	93.00	14.88	107.88
Choose	2009	6	8/14/09	141.52		16	14.15	122.00	19.52	141.52
Choose	2009	5	8/14/09	71.92		16	7.19	62.00	9.92	71.92
Choose	2009	9	8/17/09	107.88		16	10.79	93.00	14.88	107.88
Choose	2009	4	8/10/09	358.44		16	35.84	309.00	49.44	358.44
Choose	2009	1	7/28/09	719.20		16	71.92	620.00	99.20	719.20
Choose	2009	10	9/8/09	89.32	This is a JUNIT test	16	8.93	77.00	12.32	89.32

There are 8 objects in list ([hide them](#))

Figura 9.2 La lista para buscar facturas no muestra los datos del cliente

Obviamente, es difícil buscar una factura sin ver de qué cliente es. Añadamos pues el cliente a la lista usando el atributo `properties` de `@Tab` en la entidad `Invoice`, tal como muestra el listado 9.2.

Listado 9.2 Definición de datos tabulares para Invoice

```
@Tabs({
    @Tab(
        baseCondition = "deleted = false",
        properties="year, number, date, customer.number, customer.name, " +
            "vatPercentage, estimatedProfit, baseAmount, " +
            "vat, totalAmount, amount, remarks"),
    @Tab(name="Deleted", baseCondition = "deleted = true")
})
public class Invoice extends CommercialDocument {
```

Los datos tabulares por defecto (es decir, el modo lista) para una entidad son todas sus propiedades planas, pero no incluyen las referencias. Para cambiar la forma en que los datos tabulares se muestran puedes usar `properties` en la anotación `@Tab`. Estas propiedades pueden ser calificadas, es decir puedes, usando la notación del punto, poner una propiedad de una referencia, como `customer.number` y `customer.name` en este caso.

Choose a new value for Factura

	Year	Number	Date	Number of Customer	Name of Customer	VAT %	Estimated profit	Base amount	V.A.T.	Total amount
	=	=	=	=	starts	=	=			
Choose	2009	7	8/14/09	1	FRANCISCO JAVIER PANIZA LUCAS	16	86.30	744.00	119.04	863.04
Choose	2009	8	8/14/09	2	M ^{rs} CARMEN GIMENO ALABAU	16	10.79	93.00	14.88	107.88
Choose	<code>@Tab(properties="year, number, date, customer.number, customer.name, " + "vatPercentage, estimatedProfit, baseAmount, " + "vat, totalAmount, amount, remarks")</code>									141.52

Se pueden ver los datos del cliente

Figura 9.3 Gracias a `@Tab` los datos del cliente se ven en la lista de facturas

Ahora la lista para escoger una factura de un pedido es como la que se muestra en la figura 9.3.

Con esta lista de facturas es más fácil escoger la correcta, porque ahora el usuario puede ver el cliente de cada factura. Además, el usuario puede filtrar por cliente para mostrar las facturas del cliente que está buscando. Sin embargo, sería aun mejor si solo se mostraran las facturas cuyo cliente es el mismo que del pedido actual. De esta manera no habría opción para equivocarse. Lo haremos así en la siguiente sección.

9.1.3 Refinar la acción para buscar una referencia con una lista

Actualmente cuando el usuario busca una factura todas las facturas están disponibles para escoger. Vamos a mejorar esto para mostrar solo las facturas del cliente del pedido visualizado, tal como muestra la figura 9.4.

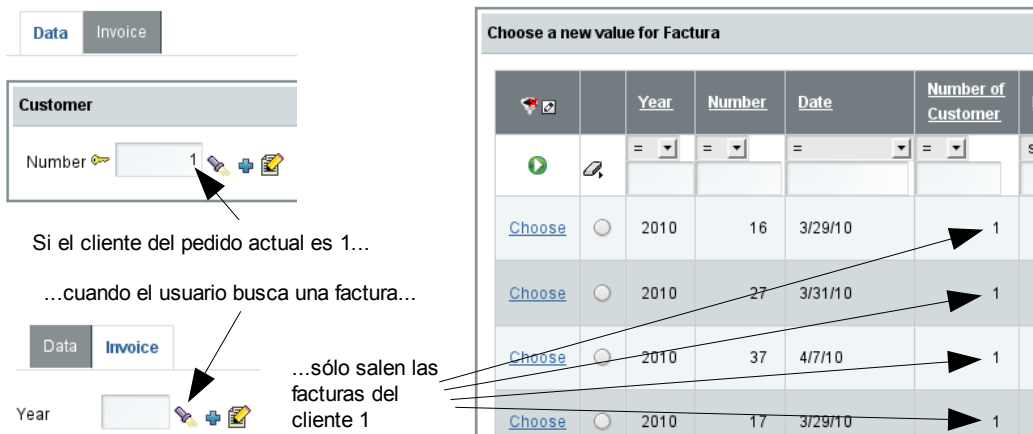


Figura 9.4 Buscar la factura desde el pedido tiene que filtrarse por cliente

Para definir nuestra propia acción de búsqueda para la referencia a factura usaremos la anotación `@SearchAction`. El listado 9.3 muestra la modificación necesaria en la clase `Order`.

Listado 9.3 `@SearchAction` define la acción personalizada para buscar facturas

```
public class Order extends CommercialDocument {

    @ManyToOne
    @ReferenceView("NoCustomerNoOrders")
    @OnChange>ShowHideCreateInvoiceAction.class
    @SearchAction("Order.searchInvoice") // Define nuestra acción para buscar facturas
    private Invoice invoice;

    ...

}
```

De esta forma tan simple definimos la acción a ejecutar cuando el usuario pulsa en el botón de la linterna para buscar una factura. El argumento usado para `@SearchAction`, `Order.searchInvoice`, es el nombre calificado de la acción, es decir la acción `searchInvoice` del controlador `Order` definido en el archivo `controllers.xml`.

Ahora tenemos que editar `controllers.xml` y añadir la definición de nuestra nueva acción, tal como muestra el listado 9.4.

Listado 9.4 Declaración de la acción `Order.searchInvoice` en `controllers.xml`

```
<controller name="Order">

    ...

    <action name="searchInvoice"
            class="org.openxava.invoicing.actions.SearchInvoiceFromOrderAction"
            hidden="true" image="images/search.gif"/>

</controller>
```

```

<!--
hidden="true": Para que no se muestre en la barra de botones del módulo
image="images/search.gif": La misma imagen que la de la acción estándar
-->

</controller>

```

Nuestra acción hereda de `ReferenceSearchAction` y ésta los necesita. El listado 9.5 muestra el código de la acción.

Listado 9.5 Acción personalizada para buscar una factura desde un pedido

```

package org.openxava.invoicing.actions; // En el paquete 'actions'

import org.openxava.actions.*; // Para usar ReferenceSearchAction

public class SearchInvoiceFromOrderAction
    extends ReferenceSearchAction { // Lógica estándar para buscar una referencia

    public void execute() throws Exception {
        super.execute(); // Ejecuta la lógica estándar, la cual muestra un diálogo
        int customerNumber =
            getPreviousView() // getPreviousView() es la vista principal (getView() es el diálogo)
                .getValueInt("customer.number"); // Lee de la vista el número
                                                // de cliente del pedido actual
        if (customerNumber > 0) { // Si hay cliente los usamos para filtrar
            getTab().setBaseCondition("${customer.number} = " + customerNumber);
        }
    }
}

```

Observa como usamos `getTab().setBaseCondition()` para establecer una condición en la lista para escoger la referencia. Es decir, desde una `ReferenceSearchAction` puedes usar `getTab()` para manipular la forma en que se comporta la lista.

Si no hay cliente no añadimos ninguna condición por tanto se mostrarían todas las facturas, esto ocurre cuando el usuario escoge la factura antes que el cliente.

9.1.4 Buscar la referencia tecleando en los campos

La lista para escoger una referencia ya funciona bien. Sin embargo, queremos dar al usuario la opción de escoger una factura sin usar la lista, simplemente tecleando el año y el número. Muy útil si el usuario conoce de antemano que factura quiere.

OpenXava provee esa funcionalidad por defecto. Si los campos `@Id` son visualizados en la referencia serán usados para buscar, en caso contrario OpenXava usa el primer campo visualizado para buscar. Aunque en nuestro caso esto no es tan conveniente, porque el primer campo visualizado es el año, y buscar una factura sólo por el año no es muy preciso. La figura 9.5 muestra el

comportamiento por defecto junto con una alternativa más conveniente.

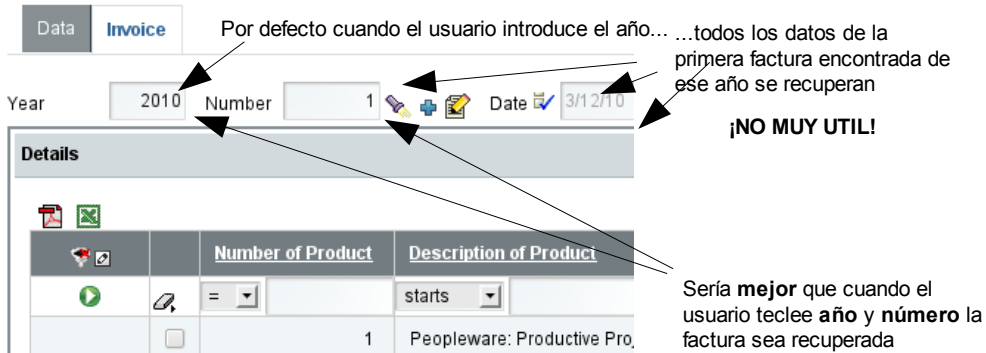


Figura 9.5 Por defecto la factura se recupera solo por año

Afortunadamente es fácil indicar que campos queremos usar para buscar desde la perspectiva del usuario. Esto se hace por medio de la anotación `@SearchKey`. Edita la clase `CommercialDocument` (recuerda, el padre de `Order` e `Invoice`) y añade esta anotación a las propiedades `year` y `number` (listado 9.6).

Listado 9.6 Definir `year` y `number` como `@SearchKey` en `CommercialDocument`

```
abstract public class CommercialDocument extends Deletable {

    ...

    @SearchKey // Añade esta anotación aquí
    @Column(length=4)
    @DefaultValueCalculator(CurrentYearCalculator.class)
    private int year;

    @SearchKey // Añade esta anotación aquí
    @Column(length=6)
    @ReadOnly
    private int number;

    ...

}
```

De esta forma cuando el usuario busque un pedido o una factura desde una referencia tiene que teclear el año y el número, y la entidad correspondiente será recuperada de la base de datos y rellenará la interfaz de usuario.

Ahora es fácil para el usuario escoger una factura desde un pedido sin usar la lista de búsqueda, simplemente tecleando el año y el número.

9.1.5 Refinar la acción para buscar cuando se teclea la clave

Ahora que obtener una factura tecleando el año y el número funciona

queremos refinarlo para ayudar al usuario a hacer su trabajo de forma más eficiente. Por ejemplo, sería útil que si el usuario todavía no ha escogido al cliente para el pedido y escoge una factura, el cliente de esa factura sea asignado automáticamente al pedido actual. La figura 9.6 visualiza el comportamiento deseado.

1. Si el pedido actual no tiene cliente

2. Cuando el usuario teclee el año y número de la factura

3. El cliente del pedido se llena con el cliente de la factura recuperada

Figura 9.6 Escoger una factura cuando todavía no hay un cliente seleccionado

Por otra parte, si el usuario ya ha seleccionado un cliente para el pedido, si no coincide con el de la factura, ésta será rechazada y se visualizará un mensaje de error, tal como muestra la figura 9.7.

Si el pedido ya tiene cliente...
...cuando el usuario escoge una factura de un cliente diferente, no se recupera y se muestra un error

Figura 9.7 Escoger una factura cuando el cliente ya está seleccionado

Para definir este comportamiento especial hemos de añadir una anotación @OnChangeSearch en la referencia invoice de Order. @OnChangeSearch permite definir nuestra propia acción para hacer la búsqueda de la referencia cuando su clave cambia en la interfaz de usuario. Puedes ver la referencia modificada en el listado 9.7.

Listado 9.7 Acción para obtener la factura desde el pedido al cambiar la clave

```

public class Order extends CommercialDocument {

    @ManyToOne
    @ReferenceView("NoCustomerNoOrders")
    @OnChange(ShowHideCreateInvoiceAction.class)
    @OnChangeSearch(OnChangeSearchInvoiceAction.class) // Añade esta anotación
    @SearchAction("Order.searchInvoice")
    private Invoice invoice;

    ...

}

```

A partir de ahora cuando un usuario teclee un nuevo año y número para la factura, `OnChangeSearchInvoiceAction` se ejecutará. En esta acción se han de leer los datos de la factura de la base de datos y actualizar la interfaz de usuario. El listado 9.8 muestra el código de la acción.

Listado 9.8 Acción para buscar la factura al teclear año y número

```

package org.openxava.invoicing.actions; // En el paquete 'actions'

import java.util.*;
import org.openxava.actions.*; // Para usar OnChangeSearchAction
import org.openxava.invoicing.model.*;
import org.openxava.model.*;
import org.openxava.view.*;

public class OnChangeSearchInvoiceAction
    extends OnChangeSearchAction { // Lógica estándar para buscar una referencia cuando
                                  // los valores clave cambian en la interfaz de usuario (1)
    public void execute() throws Exception {
        super.execute(); // Ejecuta la lógica estándar (2)
        Map keyValues = getView() // getView() aquí es la de la referencia, no la principal(3)
            .getKeyValuesWithValue();
        if (keyValues.isEmpty()) return; // Si la clave está vacía no se ejecuta más lógica
        Invoice invoice = (Invoice) // Buscamos la factura usando la clave tecleada (4)
            MapFacade.findEntity(getView().getModelName(), keyValues);
        View customerView = getView().getRoot().getSubview("customer"); // (5)
        int customerNumber = customerView.getValueInt("number");
        if (customerNumber == 0) { // Si no hay cliente lo llenamos (6)
            customerView.setValue("number", invoice.getCustomer().getNumber());
            customerView.refresh();
        }
        else { // Si ya hay un cliente verificamos que coincida con el cliente de la factura (7)
            if (customerNumber != invoice.getCustomer().getNumber()) {
                addError("invoice_customer_not_match",
                    invoice.getCustomer().getNumber(), invoice, customerNumber);
                getView().clear();
            }
        }
    }
}

```

Dado que la acción descende de `OnChangeSearchAction` (1) y usamos

`super.execute()` (2) se comporta de la forma estándar, es decir, cuando el usuario teclea el año y el número los datos de la factura se recuperan y rellenan la interfaz de usuario. Después, usamos `getView()` (3) para obtener la clave de la factura visualizada y así encontrar su correspondiente entidad usando `MapFacade` (4). Desde dentro de `ChangeSearchAction` `getView()` devuelve la subvista de la referencia, y no la vista global. Por lo tanto, en este caso `getView()` es la vista de la referencia a factura. Esto permite crear acciones `@OnChangeSearch` más reutilizables. Has de escribir `getView().getRoot().getSubview("customer")` (5) para acceder a la vista del cliente.

Para implementar el comportamiento visualizado en la anterior figura 9.6, la acción pregunta si no hay cliente (`customerNumber == 0`) (6). Si éste es el caso rellena los datos del cliente desde el cliente de la factura. En caso contrario implementa la lógica de la figura 9.7 verificando que el cliente del pedido actual coincide con el cliente de la factura recuperada.

Nos queda un pequeño detalle, el texto del mensaje. Añade la entrada mostrada en el listado 9.9 al archivo *Invoicing-messages_en.properties* de la carpeta *i18n*.

Listado 9.9 Error de búsqueda de factura en *Invoicing-messages_en.properties*

```
invoice_customer_not_match=Customer Nº {0} of invoice {1} does not match with
Customer Nº {2} of the current order
```

Una cosa interesante de `@OnChangeSearch` es que también se ejecuta si la factura se escoge desde la lista, porque en este caso el año y el número también cambian. Por ende, este es un lugar centralizado donde refinar la lógica para recuperar la referencia y rellena la vista.

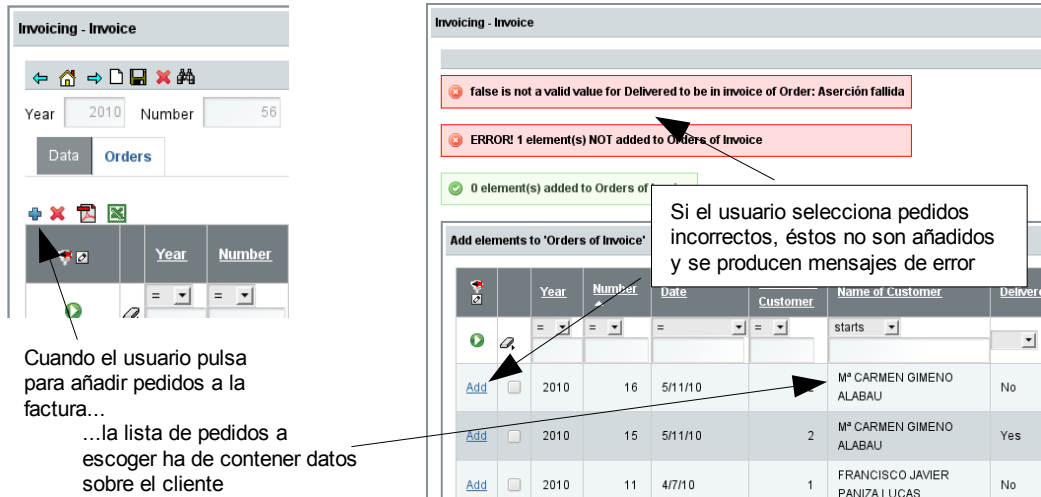
9.2 Refinar el comportamiento de las colecciones

Podemos refinar las colecciones de la misma forma que hemos hecho con las referencias. Esto es muy útil, porque nos permite mejorar el comportamiento actual del módulo *Invoice*. El usuario sólo puede añadir un pedido a una factura si la factura y el pedido pertenecen al mismo cliente. Además, el pedido tiene que estar entregado (*delivered*) y no tener todavía factura.

9.2.1 Modificar los datos tabulares ayuda

Con el comportamiento por defecto, el usuario puede tener dificultades al tratar de encontrar pedidos adecuados para asignar a su factura. Porque cuando el usuario pulsa para añadir pedidos, todos los pedidos existentes son mostrados, y lo que es peor, la información del cliente no se muestra en la lista. Queremos que los datos del cliente se muestren en la lista de pedidos, tal como muestra la figura

9.8.

**Figura 9.8 Añadir pedidos a factura mostrando los datos del cliente en la lista**

El listado 9.10 muestra como añadir el cliente a la lista usando el atributo `properties` de `@Tab` en la entidad `Order`.

Listado 9.10 Definición de datos tabulares para Order

```
@Tabs({
  @Tab(baseCondition = "deleted = false",
    properties="year, number, date, customer.number, customer.name," +
      "delivered, vatPercentage, estimatedProfit, baseAmount, " +
      "vat, totalAmount, amount, remarks"
  ),
  @Tab(name="Deleted", baseCondition = "deleted = true")
})
public class Order extends CommercialDocument {
```

Fíjate como hemos añadido `customer.number` y `customer.name`.

La figura 9.8 también muestra como la validación en las entidades impide que el usuario añada pedidos incorrectos.

Sin embargo, sería mejor si sólo los pedidos susceptibles de ser añadidos a la factura actual estuvieran presentes en la lista, de tal modo que el usuario no tuviese forma de equivocarse. Lo haremos así en la siguiente sección.

9.2.2 Refinar la lista para añadir elementos a la colección

Actualmente cuando el usuario trata de añadir pedidos a la factura todos los pedidos están disponibles. Vamos a mejorar esto para mostrar solo los pedidos del cliente de la factura, entregados y todavía sin factura, tal como muestra la figura 9.9.

Si el cliente de la factura actual es 2...

...cuando el usuario vaya a añadir pedidos...

...sólo los pedidos del cliente 2 y entregados aparecerán

Year	Number	Date	Number of Customer	Name of Customer	Delivered	VAT	Estimated profit	Base amount
2010	15	5/11/10	2	Mª CARMEN GIMENO ALABAU	Yes	16	12.41	107.00

Buttons: Add, Cancel

Figura 9.9 Añadir pedidos muestra solo los entregados y del cliente actual

Usaremos la anotación `@NewAction` para definir nuestra propia acción que muestre la lista para añadir pedidos. El listado 9.11 muestra la modificación necesaria en la clase `Order`.

Listado 9.11 `@NewAction` define la acción para ir a la lista de añadir pedidos

```
public class Invoice extends CommercialDocument {

    @OneToMany(mappedBy="invoice")
    @CollectionView("NoCustomerNoInvoice")
    @NewAction("Invoice.add0Orders") // Define nuestra propia acción para añadir pedidos
    private Collection<Order> orders;

    ...

}
```

De esta forma tan sencilla definiremos la acción a ejecutar cuando el usuario pulsa en el botón con el signo más (+) para añadir pedidos. El argumento usado para `@NewAction`, `Invoice.add0Orders`, es el nombre calificado de la acción, es decir la acción `add0Orders` del controlador `Invoice` tal como se ha definido en el archivo `controllers.xml`.

Ahora hemos de editar `controllers.xml` para añadir el controlador `Invoice` (todavía no existe) con nuestra acción. El listado 9.12 muestra la definición del controlador.

Listado 9.12 Declaración de la acción `Invoice.addOrders` en `controllers.xml`

```
<controller name="Invoice">
  <extends controller="Invoicing"/>

  <action name="add0Orders"
    class="org.openxava.invoicing.actions.GoAddOrdersToInvoiceAction"
    hidden="true" image="images/create_new.gif"/>
</controller>
```

```

<!--
hidden="true": No se mostrará en la barra de botones del módulo
image="images/create_new.gif": La misma imagen que la acción estándar
-->

</controller>

```

El listado 9.13 muestra el código de la acción.

Listado 9.13 Acción personalizada para ir a “añadir pedidos” desde una factura

```

package org.openxava.invoicing.actions; // En el paquete 'actions'

import org.openxava.actions.*; // Para usar GoAddElementsToCollectionAction

public class GoAddOrdersToInvoiceAction
    extends GoAddElementsToCollectionAction { // Lógica estándar para ir a la lista que
                                                // permite añadir elementos a la colección

    public void execute() throws Exception {
        super.execute(); // Ejecuta la lógica estándar, la cual muestra un diálogo
        int customerNumber =
            getPreviousView()// getPreviousView() es la vista principal (estamos en un diálogo)
                .getValueInt("customer.number"); // Lee el número de cliente de la
                                                // factura actual de la vista

        getTab().setBaseCondition( // La condición de la lista de pedidos a añadir
            "${customer.number} = " + customerNumber +
            " and ${delivered} = true and ${invoice.oid} is null"
        );
    }
}

```

Fíjate como usamos `getTab().setBaseCondition()` para establecer la condición de la lista para escoger la entidades a añadir. Es decir, desde una `GoAddElementsToCollectionAction` puedes usar `getTab()` para manipular la forma en que la lista se comporta.

9.2.3 Refinar la acción que añade elementos a la colección

Una mejora interesante para la colección de pedidos sería que cuando el usuario añada pedidos a la factura actual, las líneas de detalle de estos pedidos se copien automáticamente a la factura.

No podemos usar `@NewAction` para esto, porque es la acción que muestra la lista de elementos a añadir a la colección. Pero no es la acción que añade los elementos. En esta sección aprenderemos como definir la acción que realmente añade los elementos (figura 9.10).

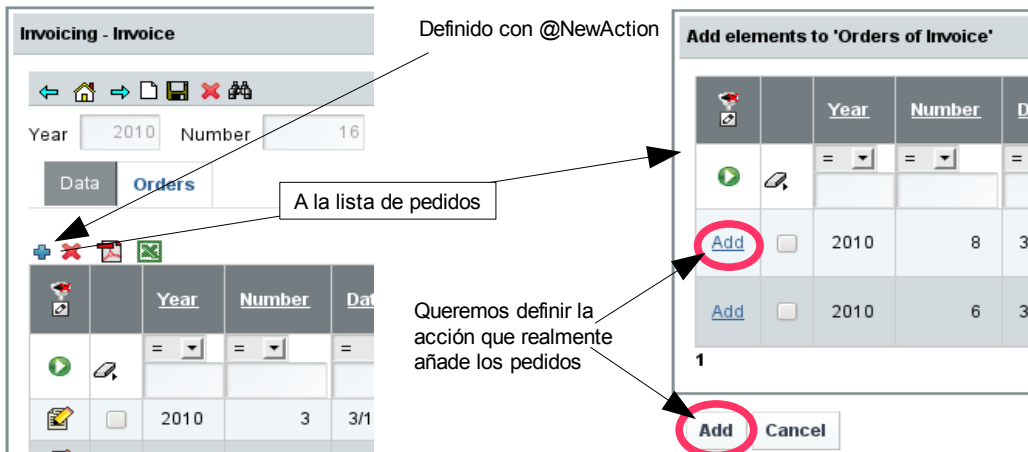


Figura 9.10 Queremos refinar la acción 'Add' en la lista de pedidos

Por desgracia, no hay una anotación para definir directamente esta acción de añadir. Sin embargo, no es una tarea demasiado difícil, solo hemos de refinar la acción `@NewAction` instruyéndola para mostrar nuestro propio controlador, y en este controlador podemos poner las acciones que queramos. Dado que ya hemos definido nuestra `@NewAction` en la sección anterior solo hemos de añadir un nuevo método a la ya existente `GoAddOrdersToInvoiceAction`. El listado 9.14 muestra este método.

Listado 9.14 `getNextController()` añadido a `GoAddOrdersToInvoiceAction`

```
public class GoAddOrdersToInvoiceAction ... {
    ...

    public String getNextController() { // Añadimos este método
        return "AddOrdersToInvoice"; // El controlador con las acciones disponibles en
    }                                     // la lista de pedidos a añadir
}
```

Por defecto las acciones en la lista de entidades a añadir (los botones 'Add' y 'Cancel') son del controlador estándar de OpenXava `AddToCollection`. Sobrescribir `getNextController()` en nuestra acción nos permite definir nuestro propio controlador en su lugar. El listado 9.15 muestra la definición de nuestro controlador propio para añadir elementos en `controllers.xml`.

Listado 9.15 Controlador personalizado para añadir pedidos a la factura

```
<controller name="AddOrdersToInvoice">
  <extends controller="AddToCollection"/> <!-- Extiende del controlador estándar -->

  <!-- Sobrescribe la acción para añadir -->
  <action name="add">
```

```

        class="org.openxava.invoicing.actions.AddOrdersToInvoiceAction"/>
    </controller>

```

De esta forma la acción para añadir pedidos a la factura será `AddOrdersToInvoiceAction`. Recuerda que el objetivo de nuestra acción es añadir los pedidos a la factura de la manera convencional, pero también copiar las líneas de estos pedidos a la factura. El listado 9.16 muestra el código de la acción.

Listado 9.16 Acción personalizada para añadir pedidos a una factura

```

package org.openxava.invoicing.actions; // En el paquete 'actions'

import java.rmi.*;
import java.util.*;
import javax.ejb.*;
import org.openxava.actions.*; // Para usar AddElementsToCollectionAction
import org.openxava.invoicing.model.*;
import org.openxava.model.*;
import org.openxava.util.*;
import org.openxava.validators.*;

public class AddOrdersToInvoiceAction
    extends AddElementsToCollectionAction { // Lógica estándar para añadir
        // elementos a la colección

    public void execute() throws Exception {
        super.execute(); // Usamos la lógica estándar "tal cual"
        getView().refresh(); // Para visualizar datos frescos, incluyendo los importes
        // recalculados, que dependen de las líneas de detalle
    }

    protected void associateEntity(Map keyValues) // El método llamado para asociar
        throws ValidationException, // cada entidad a la principal, en este caso para
            XavaException, ObjectNotFoundException, // asociar cada pedido a la factura
            FinderException, RemoteException
    {
        super.associateEntity(keyValues); // Ejecuta la lógica estándar (1)
        Order order = (Order) MapFacade.findEntity("Order", keyValues); // (2)
        order.copyDetailsToInvoice(); // Delega el trabajo principal en la entidad (3)
    }
}

```

Sobrescribimos el método `execute()` sólo para refrescar la vista después del proceso. Realmente, lo que nosotros queremos es refinar la lógica de asociar un pedido a la factura. La forma de hacer esto es sobrescribiendo el método `associateEntity()`. La lógica aquí es simple, después de ejecutar la lógica estándar (1) buscamos la entidad `Order` correspondiente y entonces llamamos al método `copyDetailsToInvoice()` de ese `Order`.

Obviamente, necesitamos tener un método `copyDetailsToInvoice()` en la entidad `Order`. El listado 9.17 muestra este método.

Listado 9.17 Método `copyDetailsInvoice()` en la clase `Order`

```

public class Order extends CommercialDocument {

    ...

    public void copyDetailsToInvoice() {
        copyDetailsToInvoice(getInvoice()); // Delegamos en un método ya existente
    }

}

```

Por suerte ya teníamos un método para copiar detalles desde una entidad Order a la Invoice especificada, simplemente llamamos a este método enviando la Invoice del Order.

Estas pequeñas modificaciones al comportamiento de la colección orders de Invoice son suficientes para convertir el módulo de Invoice en una herramienta efectiva para facturar clientes individualmente. Solo has de crear una factura nueva, escoger un cliente y añadir pedidos. Es incluso más fácil de usar que el modo lista del módulo Order (desarrollamos una acción para hacerlo en la sección 8.2) ya que el módulo Invoice solo se muestran los pedidos adecuados al cliente.

9.3 Pruebas JUnit

Todavía tenemos la sana costumbre de hacer un poco de código de aplicación, y después un poco de código de pruebas. Y ahora es el tiempo de escribir el código de pruebas para las nuevas características añadidas en esta lección.

9.3.1 Adaptar OrderTest

Si ejecutaras OrderTest ahora, no pasaría. Esto es porque nuestro código confía en ciertos detalles que han cambiado. Por lo tanto, hemos de modificar nuestro código de pruebas actual. Edita el método testSetInvoice() de OrderTest y aplica los cambios mostrados en el listado 9.18.

Listado 9.18 Modificaciones en el método testSetInvoice() de OrderTest

```

public void testSetInvoice() throws Exception {

    ...

    assertValue("invoice.number", "");
    assertValue("invoice.year", "");
    execute("Reference.search", // Ya no usamos la acción estándar para
        "keyProperty=invoice.year", // buscar la factura, en su lugar
    execute("Order.searchInvoice", // usamos nuestra acción personalizada (1)
        "keyProperty=invoice.number");
    execute("List.orderBy", "property=number");
}

```



```

...

// Restaurar valores
setValue("delivered", "false");
setValue("invoice.year", ""); // Ahora es necesario teclear el año
setValue("invoice.number", ""); // y el número para buscar la factura (2)
execute("CRUD.save");
assertNoErrors();
}

```

Recuerda que anotamos la referencia `invoice` en `Order` con `@SearchAction("Order.searchInvoice")` (sección 9.1.3), por tanto hemos de modificar la prueba para llamar a `Order.searchInvoice` (1) en vez de a `Reference.search`. En la sección 9.1.4 añadimos `@SearchKey` a `year` y `number` de `CommercialDocument`, por lo tanto nuestra prueba ha de indicar tanto `year` como `number` para obtener (o en este caso borrar) una factura (2). Por causa de esto último también hemos de modificar `testCreateInvoiceFromOrder()` de `OrderTest` como muestra el listado 9.19.

Listado 9.19 Modificaciones en `testCreateInvoiceFromOrder()` de `OrderTest`

```

public void testCreateInvoiceFromOrder() throws Exception {

    ...

    // Restaurar el pedido para ejecutar la prueba la siguiente vez
    setValue("invoice.year", ""); // Ahora es necesario teclear el año
    setValue("invoice.number", ""); // y el número para buscar la factura (2)
    assertValue("invoice.number", "");
    assertCollectionRowCount("invoice.details", 0);
    execute("CRUD.save");
    assertNoErrors();
}

```

Después de estos cambios `OrderTest` tiene que pasar. Sin embargo, todavía nos queda probar la nueva funcionalidad del módulo `Order`.

9.3.2 Probar `@SearchAction`

En la sección 9.1.3 usamos `@SearchAction` en la referencia `invoice` de `Order` para mostrar en la lista de búsqueda solo facturas del cliente del pedido actual. El listado 9.20 muestra la prueba de esta funcionalidad.

Listado 9.20 Probar buscar factura desde pedido en `OrderTest`

```

public void testSearchInvoiceFromOrder() throws Exception {
    execute("CRUD.new");
    setValue("customer.number", "1"); // Si el cliente es 1...
    execute("Sections.change", "activeSection=1");
    execute("Order.searchInvoice", // ...cuando el usuario pulsa para escoger una factura...
           "keyProperty=invoice.number");
    assertCustomerInList("1"); // ...sólo se muestran las facturas del cliente 1
}

```

```

execute("ReferenceSearch.cancel");
execute("Sections.change", "activeSection=0");
setValue("customer.number", "2"); // Y si el cliente es 2...
execute("Sections.change", "activeSection=1");
execute("Order.searchInvoice", // ...cuando el usuario pulsa para escoger una factura...
        "keyProperty=invoice.number");
assertCustomerInList("2"); // ...sólo se muestran las facturas del cliente 2
}

```

La parte más peliaguda es verificar la lista de facturas, este es el trabajo de `assertCustomerInList()` cuyo código puede verse en el listado 9.21.

Listado 9.21 Confirmar que los valores en la columna cliente son los esperados

```

private void assertCustomerInList(String customerNumber) throws Exception {
    assertListNotEmpty();
    int c = getListRowCount();
    for (int i=0; i<c; i++) { // Un bucle por todas las filas
        if (!customerNumber.equals(getValueInList(i, "customer.number"))) {
            fail("Customer in row " + i + // Si el cliente no es el esperado falla
                " is not of customer " + customerNumber);
        }
    }
}
}

```

Consiste en un bucle por todas la filas verificando el número de cliente.

9.3.3 Probar @OnChangeSearch

En la sección 9.1.5 usamos `@SearchAction` en la referencia invoice de Order para asignar automáticamente el cliente de la factura escogida al pedido actual cuando el usuario todavía no tiene cliente, o para verificar que el cliente de la factura y del pedido coinciden, si el pedido ya tiene cliente. El listado 9.22 muestra el método de prueba en `OrderTest`.

Listado 9.22 Probar los eventos al cambiar la factura de un pedido

```

public void testOnChangeInvoice() throws Exception {
    execute("CRUD.new"); // Estamos creando un nuevo pedido
    assertValue("customer.number", ""); // por tanto no tiene cliente todavía
    execute("Sections.change", "activeSection=1");
    execute("Order.searchInvoice", // Busca la factura usando una lista
            "keyProperty=invoice.number");

    execute("List.orderBy", "property=customer.number"); // Ordena por cliente
    String customer1Number = getValueInList(0, "customer.number"); // Memoriza...
    String invoiceYear1 = getValueInList(0, "year"); // ...los datos de la...
    String invoiceNumber1 = getValueInList(0, "number"); // ...primera factura
    execute("List.orderBy", "property=customer.number"); // Ordena por cliente
    String customer2Number = getValueInList(0, "customer.number"); // Memoriza...
    String customer2Name = getValueInList(0, "customer.name"); // ...los datos de...
    // ...la última factura

    assertNotEquals("Must be invoices of different customer",
        customer1Number, customer2Number); // Las 2 facturas memorizadas no son la misma
}

```

```

execute("ReferenceSearch.choose", "row=0"); // La factura se escoge con la lista (1)
execute("Sections.change", "activeSection=0");
assertValue("customer.number", customer2Number); // Los datos del cliente
assertValue("customer.name", customer2Name); // se rellenan automáticamente (2)

execute("Sections.change", "activeSection=1");
setValue("invoice.year", invoiceYear1); // Tratamos de poner una factura de...
setValue("invoice.number", invoiceNumber1); // ...otro cliente (3)

assertError("Customer Nº " + customer1Number + " of invoice " + // Muestra...
    invoiceYear1 + "/" + invoiceNumber1 + // ...un mensaje de error... (4)
    " does not match with Customer Nº " +
    customer2Number + " of the current order");

assertValue("invoice.year", ""); // ...y reinicia los datos de la factura (5)
assertValue("invoice.number", "");
assertValue("invoice.date", "");
}

```

Aquí probamos que nuestra acción on-change rellene los datos del cliente (3) al escoger una factura (2), y que si el cliente ya está establecido se muestre un mensaje de error (4) y la factura se borre de la vista (5). Fíjate como la primera vez usamos la lista (1) para escoger la factura y la segunda lo hacemos tecleando el año y el número (3).

9.3.4 Adaptar InvoiceTest

Como en el caso de OrderTest, InvoiceTest también falla. Has de hacer unos pequeños ajustes para que funcione. Edita testAddOrders() de InvoiceTest y aplica los cambios del listado 9.23.

Listado 9.23 Modificaciones en el método testAddOrders() de InvoiceTest

```

public void testAddOrders() throws Exception {
    assertListNotEmpty();
    execute("List.orderBy", "property=number");
    execute("Mode.detailAndFirst");
    execute("Sections.change", "activeSection=1");
    assertCollectionRowCount("orders", 0);
    execute("Collection.add", // La acción estándar para añadir pedidos ya no se usa
    execute("Invoice.addOrders", // En su lugar usamos nuestra propia acción
        "viewObject=xava_view_section1_orders");
    checkFirstOrderWithDeliveredEquals("Yes"); // Ahora todos los pedidos de la lista
    checkFirstOrderWithDeliveredEquals("No"); // están entregados; esto ya no hace falta

    execute("AddToCollection.add"); // En lugar de la acción estándar
    execute("AddOrdersToInvoice.add", "row=0"); // ...ahora tenemos la nuestra propia
    assertError("ERROR! 1 element(s) NOT added to Orders of Invoice"); // Es
        // imposible porque el usuario no puede escoger pedidos incorrectos
    assertMessage("1 element(s) added to Orders of Invoice");
    assertCollectionRowCount("orders", 1);
    checkRowCollection("orders", 0);
    execute("Collection.removeSelected",
        "viewObject=xava_view_section1_orders");
    assertCollectionRowCount("orders", 0);
}

```

```
}
```

Ya no necesitamos el método `checkFirstOrderWithDeliveredEquals()`, por tanto podemos quitarlo de `InvoiceTest` (listado 9.24).

Listado 9.24 Quitar `checkFirstOrderWithDeliveredEquals()` de `InvoiceTest`

```
private void checkFirstOrderWithDeliveredEquals(String value)
    throws Exception { ... }
```

Después de estos cambios `InvoiceTest` ha de funcionar. Sin embargo, todavía nos queda probar la nueva funcionalidad del módulo `Invoice`.

9.3.5 Probar `@NewAction`

En la sección 9.2.2 anotamos la colección `orders` de `Invoice` con `@NewAction` para refinar la lista de pedidos a ser añadidos a la colección. De esta forma solo los pedidos entregados del cliente de la factura actual y todavía sin facturar se mostraban. Vamos a probar esto, y al mismo tiempo, aprenderemos como refactorizar el código existente para poder reutilizarlo.

Primero queremos verificar que la lista para añadir pedidos solo contiene pedidos del cliente actual. El listado 9.25 muestra los cambios en `testAddOrders()` para conseguir esto.

Listado 9.25 Verificar que todos los pedidos en la lista son del cliente actual

```
public void testAddOrders() throws Exception {
    assertListNotEmpty();
    execute("List.orderBy", "property=number");
    execute("Mode.detailAndFirst");
    String customerNumber = getValue("customer.number"); // Tomamos nota del
    execute("Sections.change", "activeSection=1");         // cliente de la factura
    assertCollectionRowCount("orders", 0);
    execute("Invoice.addOrders",
        "viewObject=xava_view_section1_orders");
    assertCustomerInList(customerNumber); // Confirmamos que todos los cliente en
                                         // la lista coinciden con el cliente de la factura
    ...
}
```

Ahora hemos de escribir el método `assertCustomerInList()`. Pero, espera un momento, ya hemos escrito este método en `OrderTest`. Lo vimos en el listado 12.34 (sección 9.3.2). Estamos en `InvoiceTest` por tanto no podemos llamar a este método. Por fortuna tanto `InvoiceTest` como `OrderTest` heredan de `CommercialDocumentTest`, por lo tanto sólo tenemos que subir el método a la clase madre. Para hacer esto copia el método `assertCustomerInList()` desde `OrderTest` a `CommercialDocumentTest`, cambiando `private` por `protected`, tal como muestra el listado 9.26.

Listado 9.26 assertCustomerInList() movido a CommercialDocumentTest

```

abstract public class CommercialDocumentTest extends ModuleTestBase {

    private protected void // Cambiamos de private a protected
        assertCustomerInList(String customerNumber) throws Exception {

        ...

    }

    ...

}

```

Ahora puedes quitar el método `assertCustomerInList()` de `OrderTest` (listado 9.27).

Listado 9.27 assertCustomerInList() quitado de OrderTest

```

public class OrderTest extends CommercialDocumentTest {

    private void assertCustomerInList(String customerNumber)
    throws Exception { ... }

    ...

}

```

Después de estos cambios el método `testAddOrders()` compila y funciona.

No solo queremos comprobar que la lista de pedidos son del cliente correcto, sino también que están entregados. Nuestro primer impulso es copiar y pegar `assertCustomerInList()` para crear un método `assertDeliveredInList()`. Sin embargo, resistimos la tentación, y en vez de eso vamos a crear un método reutilizable. Primero, copiamos y pegamos `assertCustomerInList()` como `assertValueForAllRows()`. El listado 9.28 muestra estos dos métodos en `CommercialDocumentTest`.

Listado 9.28 Crear assertValueForAllRows() a partir de assertCustomerInList()

```

protected void assertCustomerInList(String customerNumber) throws Exception {
    assertListNotEmpty();
    int c = getListRowCount();
    for (int i=0; i<c; i++) {
        if (!customerNumber.equals(
            getValueInList(i, "customer.number"))) // Preguntamos por el cliente
                                                    // de forma fija
        {
            fail("Customer in row " + i + " is not of customer "
                + customerNumber);
        }
    }
}

protected void assertValueForAllRows(int column, String value)
    throws Exception

```

```

{
    assertListNotEmpty();
    int c = getListRowCount();
    for (int i=0; i<c; i++) {
        if (!value.equals(
            getValueInList(i, column))) // Preguntamos por la columna
            {                          // enviada como parámetro
                fail("Column " + column + " in row " + i + " is not " + value);
            }
        }
    }
}

```

Puedes ver como con unas ligeras modificaciones hemos convertido `assertCustomerInList()` en un método genérico para preguntar por el valor de cualquier columna, no solo por la del número de cliente. Ahora hemos de quitar el código redundante, puedes, bien quitar `assertCustomerInList()` o bien reimplementarlo usando el nuevo método. El listado 9.29 muestra la última opción.

Listado 9.29 Reimplementar `assertCustomerInList()` llamando al nuevo método

```

protected void assertCustomerInList(String customerNumber) throws Exception {
    assertValueForAllRows(3, customerNumber); // Número de cliente está en la columna 3
}

```

Usemos `assertValueForAllRows()` para verificar que la lista de pedidos contiene solo pedidos entregados. El listado 9.30 muestra la modificación necesaria en `testAddOrders()` de `InvoiceTest`.

Listado 9.30 Confirmar que todos los pedidos están entregados

```

public void testAddOrders() throws Exception {
    assertListNotEmpty();
    execute("List.orderBy", "property=number");
    execute("Mode.detailAndFirst");
    String customerNumber = getValue("customer.number");
    execute("Sections.change", "activeSection=1");
    assertCollectionRowCount("orders", 0);
    execute("Invoice.addOrders",
        "viewObject=xava_view_section1_orders");
    assertCustomerInList(customerNumber);
    assertValueForAllRows(5, "Yes"); // Todas las celdas de la columna 5 (delivered)
                                    // tienen 'Yes'
    ...
}

```

Además, queremos que solo los pedidos sin factura se muestren en la lista. Una forma sencilla de hacerlo es verificando que después de añadir un pedido a la factura actual, la lista de pedidos tenga una entrada menos. El listado 9.31 muestra los cambios necesarios en `testAddOrders()` para hacer esto.

Listado 9.31 Probar que los pedidos ya añadidos no pueden añadirse otra vez

```

public void testAddOrders() throws Exception {

    ...

    assertCustomerInList(customerNumber);
    assertValueForAllRows(5, "Yes");
    int ordersRowCount = getListRowCount(); // Tomamos nota de la cantidad de pedidos
    execute("AddOrdersToInvoice.add", "row=0"); // cuando se muestra la lista
    assertMessage("1 element(s) added to Orders of Invoice");
    assertCollectionRowCount("orders", 1); // Se añadió un pedido
    execute("Invoice.addOrders", // Mostramos la lista de pedidos otra vez
           "viewObject=xava_view_section1_orders");
    assertListRowCount(ordersRowCount - 1); // Tenemos un pedido menos en la lista
    execute("AddToCollection.cancel");

    ...

}

```

Con el código de esta sección hemos probado la `@NewAction` de la colección `orders`, y al mismo tiempo hemos visto como no es necesario crear código genérico desde el principio, porque no es difícil convertir el código concreto en genérico bajo demanda.

9.3.6 Probar la acción para añadir elementos a la colección

En la sección 9.2.3 aprendimos como refinar la acción para añadir pedidos a la factura, ahora es el momento de escribir su correspondiente código de prueba. Recuerda que esta acción copia las líneas de los pedidos seleccionados a la factura actual. El listado 9.32 muestra los cambios para probar nuestra acción personalizada para añadir pedidos.

Listado 9.32 Cuando un pedido se añade sus líneas se añaden a la factura

```

public void testAddOrders() throws Exception {

    ...

    String customerNumber = getValue("customer.number");
    deleteDetails(); // Borra la líneas de detalle si las hay (1)
    assertCollectionRowCount("details", 0); // Ahora la factura no tiene detalles
    assertValue("baseAmount", "0.00"); // Sin detalles el importe base es 0
    execute("Sections.change", "activeSection=1");
    assertCollectionRowCount("orders", 0);
    execute("Invoice.addOrders", // Cuando mostramos la lista de pedidos (2) ...
           "viewObject=xava_view_section1_orders");
    assertCustomerInList(customerNumber);
    assertValueForAllRows(5, "Yes");
    String firstOrderBaseAmount = getValueInList(0, 8); // ..tomamos nota del importe
    int ordersRowCount = getListRowCount(); // base del primer pedido de la lista (3)
    ...
    assertCollectionRowCount("orders", 1);
    execute("Sections.change", "activeSection=0");
    assertCollectionNotEmpty("details"); // Hay detalles, han sido copiados (4)
}

```

```

assertValue("baseAmount", firstOrderBaseAmount); // El importe base de la factura
execute("Sections.change", "activeSection=1");      // coincide con el del
...                                                  // pedido recién añadido (5)
}

```

Quitamos las líneas de detalle de la factura (1), después añadimos un pedido (2), tomando nota de su importe base (3), entonces verificamos que la factura actual tiene detalles (4) y que su importe base es el mismo que el del pedido añadido (5).

Nos queda el método `deleteDetails()`, mostrado en el listado 9.33.

Listado 9.33 Borra todos los detalles de la factura visualizada

```

private void deleteDetails() throws Exception {
    int c = getCollectionRowCount("details");
    for (int i=0; i<c; i++) { // Un bucle por todas las filas
        checkRowCollection("details", i); // Marca cada fila
    }
    execute("Collection.removeSelected", // Borra las filas marcadas
           "viewObject=xava_view_section0_details");
}

```

Selecciona todas las filas de la colección `details` y pulsa en el botón 'Remove selected'.

El método `testAddOrders()` está acabado. Puedes ver su código definitivo en el listado 9.34.

Listado 9.34 Código definitivo para `testAddOrders()` de `InvoiceTest`

```

public void testAddOrders() throws Exception {
    assertListNotEmpty();
    execute("List.orderBy", "property=number");
    execute("Mode.detailAndFirst");
    String customerNumber = getValue("customer.number");
    deleteDetails();
    assertCollectionRowCount("details", 0);
    assertValue("baseAmount", "0.00");
    execute("Sections.change", "activeSection=1");
    assertCollectionRowCount("orders", 0);
    execute("Invoice.addOrders",
           "viewObject=xava_view_section1_orders");
    assertCustomerInList(customerNumber);
    assertValueForAllRows(5, "Yes");
    String firstOrderBaseAmount = getValueInList(0, 8);
    int ordersRowCount = getListRowCount();
    execute("AddOrdersToInvoice.add", "row=0");
    assertMessage("1 element(s) added to Orders of Invoice");
    assertCollectionRowCount("orders", 1);
    execute("Sections.change", "activeSection=0");
    assertCollectionNotEmpty("details");
    assertValue("baseAmount", firstOrderBaseAmount);
    execute("Sections.change", "activeSection=1");
}

```



```

execute("Invoice.addOrders",
    "viewObject=xava_view_section1_orders");
assertListRowCount(ordersRowCount - 1);
execute("AddToCollection.cancel");
checkRowCollection("orders", 0);
execute("Collection.removeSelected",
    "viewObject=xava_view_section1_orders");
assertCollectionRowCount("orders", 0);
}

```

Hemos finalizado el código de las pruebas automáticas. Ahora puedes ejecutar todas las pruebas de tu proyecto. Han de salir en color verde.

9.4 Resumen

Esta lección te ha mostrado como refinar el comportamiento estándar de las referencias y colecciones para que tu aplicación se adapte a las necesidades del usuario. Aquí sólo has visto algunos ejemplos ilustrativos. OpenXava ofrece muchas más posibilidades para refinar el comportamiento de las colecciones y referencias, con anotaciones como `@ReferenceView`, `@ReadOnly`, `@NoFrame`, `@NoCreate`, `@NoModify`, `@NoSearch`, `@AsEmbedded`, `@SearchAction`, `@DescriptionsList`, `@LabelFormat`, `@Action`, `@OnChange`, `@OnChangeSearch`, `@Editor`, `@CollectionView`, `@EditOnly`, `@ListProperties`, `@RowStyle`, `@EditAction`, `@ViewAction`, `@NewAction`, `@SaveAction`, `@HideDetailAction`, `@RemoveAction`, `@RemoveSelectedAction`, `@ListAction`, `@DetailAction` o `@OnSelectElementAction`.

Y por si esto fuera poco, siempre tienes la opción definir tu propio editor para referencias o colecciones. Los editores te permiten crear una interfaz de usuario personalizada para visualizar y editar la referencia o colección.

Esta flexibilidad te permite usar la generación automática de la interfaz gráfica para prácticamente cualquier caso posible en las aplicaciones de gestión de la vida real.